ν-Types for Effects and Freshness Analysis

Massimo Bartoletti[1], Pierpaolo Degano[2], Gian Luigi Ferrari[2], and Roberto Zunino[3]

[1] Dipartimento di Matematica e Informatica, Università degli Studi di Cagliari, Italy
[2] Dipartimento di Informatica, Università di Pisa, Italy
[3] Dipartimento di Ingegneria e Scienza dell'Informazione, Università di Trento, Italy

Abstract. We define a type and effect system for a λ-calculus extended with side effects, in the form of primitives for creating and accessing resources. The analysis correctly over-approximates the sequences of resource accesses performed by a program at run-time. To accurately analyse the binding between the creation of a resource and its accesses, our system exploits a new class of types. Our ν-types have the form $\nu N.\, \tau \triangleright H$, where the names in N are bound both in the type τ and in the effect H, that represents the sequences of resource accesses.

1 Introduction

The paramount goal of static analysis is that of constructing sound, and as precise as possible, approximations to the behaviour of programs. Various kinds of behaviour have been studied, to guarantee that the analysed programs enjoy some properties of interest: for instance, that a program has no type errors, that communication channels are used correctly, that the usage of resources respects some prescribed policy, *etc.* In the classical approach to type systems, one approximates values and expressions as types, and at the same time checks the desired property over the constructed abstraction.

Separating the concerns of constructing the approximation and of verifying it has some advantages, however. First, once the first step is done, one can check the same abstract behaviour against different properties. Second, one can independently improve the accuracy of the first analysis and the efficiency of the verification algorithm. Third, if we devise a complete verification technique (for a given abstraction), then we have a good characterization of the accuracy of the abstraction with respect to the property of interest.

In this paper, we propose a new sort of types (called *ν-types*) for classifying programs according to their abstract behaviour, that we define as follows. Call *resource* any program object (a variable, a channel, a kernel service, *etc.*) relevant for the property of interest, and call *event* any action performed on a resource (a variable assignment, an output along a channel, a system call, *etc.*). Then, the abstract behaviour we are concerned with is the set of all the possible sequences of events (*histories*) that can result from the execution of a program.

Our reference program model is a call-by-value λ-calculus extended with side effects, that model events, and with a primitive for creating new resources. Our

ν-types have the form $\nu N.\, \tau \rhd H$, where the names $n \in N$ are bound both in the type τ and in the effect H, that is a *history expression* that represents the possible histories. Essentially, history expressions are Basic Process Algebra [7] processes extended with name restriction *à la* π-calculus [14]. We showed in [5] that history expressions are a suitable model upon which one can develop sound and complete techniques for verifying history-based usage policies of programs.

The possibility of creating new resources poses the non-trivial problem of correctly recording the binding of a fresh name with its possible uses in types and effects. For instance, consider the following function:

$$f = \lambda y.\, \mathbf{new}\ x\ \mathbf{in}\ \alpha(x); x$$

Each application of f creates a new resource r, fires the event $\alpha(r)$, and finally returns r. A suitable ν-type for f would then be $(\mathbf{1} \to (\nu n.\{n\} \rhd \alpha(n))) \rhd \varepsilon$. The unit type $\mathbf{1}$ for the parameter y is irrelevant here. Since f is a function, the actual effect is empty, denoted by the history expression ε. The return type $\nu n.\, \{n\} \rhd \alpha(n)$ correctly predicts the behaviour of applying f. The binder νn guarantees the freshness of the name n in the type $\{n\}$ – which indicates that f will return a fresh resource r – and in the history expression $\alpha(n)$. Indeed, $\nu n.\, \alpha(n)$ abstracts from any sequence $\alpha(r)$, where r is a fresh resource.

Consider now the following term:

$$\mathbf{let}\ f = \lambda y.\, \mathbf{new}\ x\ \mathbf{in}\ \alpha(x); x\ \ \mathbf{in}\ \ \beta(f*; f*)$$

Here we apply f twice to the value $*$, and we fire β on the resource that results from the second application of f. A suitable ν-type for the above would be:

$$\mathbf{1} \rhd (\nu n.\ \alpha(n)) \cdot (\nu n'.\ \alpha(n') \cdot \beta(n'))$$

The first part $\nu n.\ \alpha(n)$ of the history expression describes the behaviour of the first application of f, while the second part $\nu n'.\ \alpha(n') \cdot \beta(n')$ approximates the second application, and firing β on the returned name n'. The binders ensure that the resources represented by n and n' are kept distinct.

As a more complex example, consider the following recursive function (where z stands for the whole function g within its body):

$$g = \lambda_z x.\, \mathbf{new}\ y\ \mathbf{in}\ (\alpha(y); (b(x))\,?\, x : (b'(y))\,?\, z\, y : z\, x)$$

The function g creates a new resource upon each loop; if g ever terminates, it either returns the resource passed as parameter, or one of the resources created. If no further information is known about the boolean predicates b and b', we cannot statically predict which resource is returned. A suitable ν-type for g is:

$$(\{?\} \to (\{?\} \rhd \mu h.\, \nu n.\, \alpha(n) \cdot (h + \varepsilon))) \rhd \varepsilon$$

Being g a function, its actual effect is ε. Its functional type is $\{?\} \to \{?\}$, meaning that g takes as parameter any resource, and it returns an unknown resource. The latent effect $\mu h.\, \nu n.\, \alpha(n) \cdot (h + \varepsilon)$ represents the possible histories generated when applying g, i.e. any finite sequence $\alpha(r_0) \cdots \alpha(r_k)$ such that $r_i \neq r_j$ for all $i \neq j$.

The examples given above witness some inherent difficulties of handling new names in static analysis. We take as starting point the type and effect system of [18], which handles a λ-calculus with side effects, but without resource creation. We extend the calculus of [18] with the **new** primitive, and we give it a big-step operational semantics. We then define effects (i.e. history expressions) and our ν-types, together with a subtyping/subeffecting relation. We introduce then a type and effect system for our calculus, which associates any well-typed term with a ν-type that correctly approximates the possible run-time histories. We finally present some possible extensions to our work. Further typing examples and the proofs of our statements can be found in [6].

Related work. Our investigation started in [1] to deal with history-based access control in a calculus with side effects, but without creation of resources. In a subsequent paper [3] we featured a preliminary treatment of resource creation, through a conservative extension of simple types. The idea was that of using a special event $new(n)$ as a "weak" binder – a sort of *gensym()* – instead of using explicit ν-binders. While this allowed for reusing some of the results of [18], e.g. type inference, it also required a further analysis step, called "bindification" to place the ν-binders at the right points in the inferred effect. A first drawback of this approach is that bindification is not always defined, because the introduced scopes of names may interfere dangerously, e.g. in $new(n) \cdot new(n) \cdot \alpha(n)$. A second, more serious, drawback is that our theory of weak binders resulted too complex to be usable in practice [4]. Several definitions (e.g. the bound and free names, the semantics of history expressions, and the subeffecting relation) needed particular care to deal with the corner cases, so leading to extremely intricate proofs. The ν-types presented here are an attempt to solve both these problems. For the first problem, bindification is no longer needed, because ν-binders are already embodied into types. For the second problem, we found the proofs about ν-types, although not immune from delicate steps (e.g. checking capture avoidance in α-conversions) are far easier than those with weak binders. Another technical improvement over [3] is the Subject Reduction Lemma. Actually, in [3] we used a small-step semantics, which "consumes" events as they are fired. As a consequence, the effect of an term cannot be preserved under transitions. To prove type soundness, we had then to deal with a weak version of Subject Reduction, where the effects before and after a transition are in a somewhat convoluted relation. The proof of this statement was extremely complex, because of the weak induction hypothesis. Unlike [3], here we adopt a big-step semantics, which does not consume events. This allows us to establish Subject Reduction in the classical form, where the type is preserved under transitions.

In [2] we combined a type and effect analysis and a model-checking technique in a unified framework, to statically verify history-based policies of programs, in a λ-calculus enriched with primitives to create and use resources, and lexically-scoped usage policies. The present paper extends some results of [2] by presenting further technical achievements about the type and effect system and its relation with the program semantics, in a cleaner setting.

A number of formal techniques have been developed to handle binding and freshness of names. The language FreshML [17] has constructors and destructors for handling bound names. This allows for elegantly manipulating object-level syntactical structures up-to α-conversion, so relieving programmers from the burden of explicitly handling capture-avoidance. The FreshML type system however has a different goal than ours, since it extends the ML type system, while it is not concerned with approximating run-time histories like ours.

Skalka and Smith [18,19] proposed a λ-calculus with local checks that enforce linear μ-calculus properties [8] on the past history. A type and effect system approximates the possible run-time histories, whose validity can be statically verified by model checking μ-calculus formulae over Basic Process Algebras [7,10]. Compared with our type system, [18] also allows for let-polymorphism, subtyping of functional types, and type inference – but it does not handle resource creation. In Sec. 5 we further discuss these issues.

Regions have been used in type and effect systems [20,15] to approximate new names in impure call-by-value λ-calculi. The static semantics of [15], similarly to ours, aims at over-approximating the set of run-time traces, while that of [20] only considers flat sets of events. A main difference from our approach is that, while our ν-types deal with the *freshness* of names, both [20] and [15] use universal polymorphism for typing resource creations. Since a region n stands for a *set* of resources, in an effect $\alpha(n) \cdot \beta(n)$ their static approximation does not ensure that α and β act on the same resource. This property can instead be guaranteed in our system through the effect $\nu n.(\alpha(n) \cdot \beta(n))$. This improvement in the precision of approximations is crucial, since it allows us to model-check in [5] regular properties of traces (e.g. permit *read*(*file*) only after an *open*(*file*)) that would otherwise fail with the approximations of [20,15].

Igarashi and Kobayashi [12] extended the λ-calculus with primitives for creating and accessing resources, and for defining their permitted usage patterns. An execution is resource-safe when the possible patterns are within the permitted ones. A type system guarantees well-typed expressions to be resource-safe. Types abstract the usages permitted at run-time, while typing rules check that resource accesses respect the deduced permitted usages. Since the type system checks resource-safety while constructing the types, type inference is undecidable in the general case. Separating the analysis of effects from their verification, as we did here, led to a simpler model of types. Also, it allowed us to obtain in [5] a sound, complete and PTIME verification algorithm for checking approximations against usage policies. Clearly, also [12] would be amenable to verification, provided that one either restricts the language of permitted usages to a decidable subset, or one uses a sound but incomplete algorithm.

The $\lambda\nu$-calculus of [16] extends the pure λ-calculus with names. In contrast to λ-bound variables, nothing can be substituted for a name, yet names can be tested for equality. Reduction is confluent, and it allows for deterministic evaluation; also, all the observational equivalences of the pure λ-calculus still hold in $\lambda\nu$. Unlike our calculus, names cannot escape their static scope, e.g. $\nu n.n$ is stuck. Consequently, the type system of $\lambda\nu$ is not concerned with name extrusion (and approximation of traces), which is a main feature of ours.

Types and effects are also successfully used in process calculi. Honda, Yoshida and Carbone [11] defined multi-party session types to ensure a correct orchestration of complex systems. Unlike ours, their types do not contain ν binders: the main feature there is not tracking name flow, but reconciling global and local views of multi-party protocols. Igarashi and Kobayashi [13] and Chaki, Rajamani and Rehof [9] defined behavioural types for the π-calculus. In both these proposals, a π-calculus process is abstracted into a CCS-like processes, with no operators for hiding or creating names. Abstractions with ν-binders, however, make it possible to statically verify relevant usage properties about the fresh resources used by a program (see e.g. [5]).

2 A Calculus for Resource Access and Creation

In our model, *resources* are system objects that can either be statically available in the environment (Res_s, a finite set), or be dynamically created (Res_d, a denumerable set). Resources are accessed through a given finite set of *actions*. An *event* $\alpha(r)$ abstracts from accessing the resource r through the action α. When the target resource of an action α is immaterial, we stipulate that α acts on some special (static) resource, and we write just α for the event. A *history* is a finite sequence of events. In Def. 1 we introduce the needed syntactic categories.

Definition 1. Syntactic categories

$r, r', \ldots \in \mathsf{Res} = \mathsf{Res}_s \cup \mathsf{Res}_d$	*resources (static/dynamic)*
$\alpha, \alpha', \ldots \in \mathsf{Act}$	*actions (a finite set)*
$\alpha(r), \ldots \in \mathsf{Ev} = \mathsf{Act} \times \mathsf{Res}$	*events* ($\eta, \eta', \ldots \in \mathsf{Ev}^*$ *are histories)*
$x, x', \ldots \in \mathsf{Var}$	*variables*
$n, n', \ldots \in \mathsf{Nam}$	*names*

We consider an impure call-by-value λ-calculus with primitives for creating and accessing resources. The syntax is in Def. 2. Variables, abstractions, applications and conditionals are as expected. The definition of guards b in conditionals is irrelevant here, and so it is omitted. The variable z in $\lambda_z x.\, e$ is bound to the whole abstraction, so to allow for an explicit form of recursion. The parameter of an event may be either a resource or a variable. The term **new** represents the creation of a fresh resource. The term ***!*** models an aborted computation.

Definition 2. Syntax of terms

$e, e' ::= x$	*variable*	
r	*resource*	
$(b)\,?\,e : e'$	*conditional*	
$\lambda_z x.\, e$	*abstraction*	$(x, z \in \mathsf{Var})$
$e\, e'$	*application*	
$\alpha(\xi)$	*event*	$(\xi \in \mathsf{Var} \cup \mathsf{Res})$
new	*resource creation*	
!	*aborted computation*	

Values $v, v', \ldots \in \mathsf{Val}$ are variables, resources, abstractions, and the term $\boldsymbol{!}$. We write $*$ for a fixed, closed value. We shall use the following abbreviations, the first four of which are quite standard:

$$\begin{array}{ll}
\lambda_z.\, e = \lambda_z x.\, e \quad \text{if } x \notin \mathit{fv}(e) & \lambda x.\, e = \lambda_z x.\, e \quad \text{if } z \notin \mathit{fv}(e) \\
e; e' = (\lambda.\, e')\, e & (\mathbf{let}\ x = e\, \mathbf{in}\ e') = (\lambda x.\, e')\, e \\
\mathbf{new}\ x\ \mathbf{in}\ e = (\lambda x.\, e)\, (\mathbf{new}) & \alpha(e) = (\mathbf{let}\ z = e\, \mathbf{in}\ \alpha(z))
\end{array}$$

Some auxiliary notions are needed to define the operational semantics of terms. A *history context* is a finite representation of an infinite set of histories that only differ for the choice of fresh resources. For instance, the set of histories $\{\, \alpha(r) \mid r \in \mathsf{Res} \,\}$ is represented by the context $\mathbf{new}\ x\ \mathbf{in}\ \alpha(x); \bullet$. Contexts composition is crucial for obtaining compositionality.

Definition 3. History contexts

A history context C *is inductively defined as follows:*

$$C \ ::= \ \bullet \ \mid \ \alpha(\xi);\, C \ \mid \ \mathbf{new}\ x\, \mathbf{in}\ C$$

The free and the bound variables $\mathit{fv}(C)$ and $\mathit{bv}(C)$ of C are defined as expected. We write $C[C']$ for $C[C'[\bullet]]$, also assuming the needed α-conversions of variables so to ensure $\mathit{bv}(C) \cap \mathit{bv}(C') = \emptyset$ (note that $\mathit{bn}(C) \cap \mathit{fn}(C') \neq \emptyset$ is ok).

We specify in Def. 4 our operational semantics of terms, in a big-step style. Transitions have the form $e \overset{C}{\Longrightarrow} v$, meaning that the term e evaluates to the value v, while producing a history denoted by C.

Definition 4. Big-step semantics of terms

The big-step semantics of a term e is defined by the relation $e \overset{C}{\Longrightarrow} v$, which is the least relation closed under the rules below.

$$\textsc{E-Val}\ \ v \overset{\bullet}{\Longrightarrow} v \qquad \textsc{E-Bang}\ \ e \overset{\bullet}{\Longrightarrow} \boldsymbol{!} \qquad \textsc{E-If}\ \frac{e_{\mathcal{B}(b)} \overset{C}{\Longrightarrow} v}{(b)\, ?\, e_{\mathit{tt}} : e_{\mathit{ff}} \overset{C}{\Longrightarrow} v}$$

$$\textsc{E-Ev}\ \ \alpha(\xi) \xrightarrow{\alpha(\xi);\,\bullet} * \qquad \textsc{E-New}\ \ \mathbf{new} \xrightarrow{\mathbf{new}\ x\ \mathbf{in}\ \bullet} x$$

$$\textsc{E-Beta}\ \frac{e \overset{C}{\Longrightarrow} \lambda_z x.\, e'' \quad e' \overset{C'}{\Longrightarrow} v' \neq \boldsymbol{!} \quad e''\{v'/x, \lambda_z x.\, e''/z\} \overset{C''}{\Longrightarrow} v}{e\, e' \xrightarrow{C[C'[C'']]} v}$$

$$\textsc{E-BetaBang1}\ \frac{e \overset{C}{\Longrightarrow} \boldsymbol{!}}{e\, e' \overset{C}{\Longrightarrow} \boldsymbol{!}} \qquad \textsc{E-BetaBang2}\ \frac{e \overset{C}{\Longrightarrow} v \neq \boldsymbol{!} \quad e' \overset{C'}{\Longrightarrow} \boldsymbol{!}}{e\, e' \xrightarrow{C[C']} \boldsymbol{!}}$$

The rules (E-Val) and (E-Ev) are straightforward. The rule (E-Bang) aborts the evaluation of a term, so allowing us to observe the finite prefixes of its histories. For conditionals, the rule (E-If) assumes as given a total function $\mathcal{B}$ that evaluates the boolean guards. The rule (E-New) evaluates a **new** to a variable x, and records in the context **new** x **in** $\bullet$ that x may stand for any (fresh) resource. The last three rules are for β-reduction of an application $e\,e'$. The rule (E-Beta) is used when both the evaluations of e and e' terminate; (E-BetaBang1) is for when the evaluation of e has been aborted; (E-BetaBang2) is used when the evaluation e terminates while that of e' has been aborted.

Example 1. Let $e = (\lambda y.\, \alpha(y))\,\mathbf{new}$. We have that:

$$\frac{\lambda y.\, \alpha(y) \xRightarrow{\bullet} \lambda y.\, \alpha(y) \qquad \mathbf{new} \xRightarrow{\mathbf{new}\ x\ \mathbf{in}\ \bullet} x \qquad \alpha(x) \xRightarrow{\alpha(x);\bullet} *}{e \xRightarrow{\mathbf{new}\ x\ \mathbf{in}\ \alpha(x);\bullet} *}$$

Consider now the following two recursive functions:

$$f = \lambda_z x.\, (\alpha; zx) \qquad g = \lambda_z x.\, \mathbf{new}\ y\ \mathbf{in}\ (b(x))\,?\,y : z*$$

The function f fires the event α and recurse. The function g creates a new resource upon each loop; if it ever terminates, it returns the last resource created. For all $k \geq 0$ and for all contexts C, let C^k be inductively defined as $C^0 = \bullet$ and $C^{k+1} = C[C^k]$. Then, for all $k \geq 0$, we have that $f* \xRightarrow{(\alpha;\bullet)^k} \boldsymbol{!}$, and, assuming $b(x)$ non-deterministic, $g* \xRightarrow{(\mathbf{new}\ w\ \mathbf{in}\ \bullet)^k} \boldsymbol{!}$ and $g* \xRightarrow{(\mathbf{new}\ w\ \mathbf{in}\ \bullet)^k[\mathbf{new}\ y\ \mathbf{in}\ \bullet]} y$. □

We now define the set of histories $\mathcal{H}(e)$ that a term e can produce at run-time. To this purpose, we exploit the auxiliary operator $\mathcal{H}(C, R)$, that constructs the set of histories denoted by the context C under the assumption that R is the set of available resources (Def. 5). Note that all the histories in $\mathcal{H}(e)$ are "truncated" by a !. Only looking at $\mathcal{H}(e)$, gives then no hint about the termination of e. However, this is not an issue, since our goal is not checking termination, but approximating all the possible histories a term can produce.

Definition 5. Run-time histories

For each history context C such that $fv(C) = \emptyset$, for all $R \subseteq \mathsf{Res}$, and for all terms e, we define $\mathcal{H}(C, R)$ and $\mathcal{H}(e)$ inductively as follows:

$$\begin{aligned}
\mathcal{H}(\bullet, R) &= \{\,!\,\} \\
\mathcal{H}(\alpha(r); C, R) &= \{\,!\,\} \cup \{\, \alpha(r)\eta \mid \eta \in \mathcal{H}(C, R) \,\} \\
\mathcal{H}(\mathbf{new}\ x\ \mathbf{in}\ C, R) &= \{\,!\,\} \cup \textstyle\bigcup_{r \notin R \cup \mathsf{Res}_s} \mathcal{H}(C\{r/x\}, R \cup \{r\}) \\
\mathcal{H}(e) &= \{\, \eta \in \mathcal{H}(C, \emptyset) \mid e \xRightarrow{C} v \,\}
\end{aligned}$$

Example 2. Recall from Ex. 1 the term $e = (\lambda y.\, \alpha(y))\,\mathbf{new}$. All the possible observations (i.e. the histories) of the runs of e are represented by $\mathcal{H}(e) = \mathcal{H}(\mathbf{new}\ x\ \mathbf{in}\ \alpha(x); \bullet, \emptyset) = \{\,!\,\} \cup \bigcup_{r \in \mathsf{Res}} \{\alpha(r)\,!\}$. Note how the variable x in C was instantiated with all the possible fresh resources r. □

3 Effects and Subeffecting

History expressions are used to approximate the behaviour of terms. They include ε, representing the empty history, variables h, events $\alpha(\rho)$, resource creation $\nu n.H$, sequencing $H \cdot H'$, non-deterministic choice $H + H'$, recursion $\mu h.H$, and $!$, a nullary event that models an aborted computation. Hereafter, we assume that actions can also be fired on a special, unknown resource denoted by "?", typically due to approximations made by the type and effect system. In $\nu n.\, H$, the free occurrences of the name n in H are bound by ν; similarly acts μh for the variable h. The free variables $fv(H)$ and the free names $fn(H)$ are defined as expected. A history expression H is *closed* when $fv(H) = \emptyset = fn(H)$.

Definition 6. Syntax of history expressions

$H, H' ::=$	ε	*empty*	
	$!$	*truncation*	
	h	*variable*	
	$\alpha(\rho)$	*event*	$(\rho \in \mathsf{Res} \cup \mathsf{Nam} \cup \{?\})$
	$\nu n.H$	*resource creation*	
	$H \cdot H'$	*sequence*	
	$H + H'$	*choice*	
	$\mu h.H$	*recursion*	

We define below a denotational semantics of history expressions. Compared with [18,3], where labelled transition semantics were provided, here we find a denotational semantics more suitable, e.g. for reasoning about the composition of effects. Some auxiliary definitions are needed.

The binary operator $\odot$ (Def. 7) composes sequentially a history η with a set of histories X, while ensuring that all the events after a $!$ are discarded. For instance, $H = (\mu h.\, h) \cdot \alpha(r)$ will never fire the event $\alpha(r)$, because of the infinite loop that precedes the event. In our semantics, the first component $\mu h.\, h$ will denote the set of histories $\{\,!\,\}$, while $\alpha(r)$ will denote $\{\,!, \alpha(r), \alpha(r)\,!\,\}$. Combining the two semantics results in $\{\,!\,\} \odot \{\,!, \alpha(r), \alpha(r)\,!\,\} = \{\,!\,\}$.

Definition 7. *Let $X \subseteq \mathsf{Ev}^* \cup \mathsf{Ev}^*\,!$, and $x \in \mathsf{Ev} \cup \{\,!\,\}$. We define $x \odot X$ and its homomorphic extension $\eta \odot X$, where $\eta = a_1 \cdots a_n$, as follows:*

$$x \odot X = \begin{cases} \{\, x\eta \mid \eta \in X \,\} & \text{if } x \neq ! \\ \{x\} & \text{if } x = ! \end{cases} \qquad \eta \odot X = a_1 \odot \cdots \odot a_n \odot X$$

The operator $\boxdot$ (Def. 8) defines sequential composition between semantic functions, i.e. functions from (finite) sets of resources to sets of histories. To do that, it records the resources created, so to avoid that a resource is generated twice. For instance, let $H = (\nu n.\, \alpha(n)) \cdot (\nu n'.\, \alpha(n'))$. The component $\nu n'.\, \alpha(n')$ must not generate the same resources as the component $\nu n.\, \alpha(n)$, e.g. $\alpha(r_0)\alpha(r_0)$ is *not* a possible history of H. The definition of $\boxdot$ exploits the auxiliary function R, that singles out the resources occurring in a history η. Also, $\downarrow \in \mathsf{R}(\eta)$ indicates that η is terminating, i.e. it does not contain any $!$'s denoting its truncation.

Definition 8. *Let* $Y_0, Y_1 : \mathcal{P}_{fin}(\mathsf{Res}) \to \mathcal{P}(\mathsf{Ev}^* \cup \mathsf{Ev}^*\,!)$. *The composition* $Y_0 \boxdot Y_1$ *is defined as follows:*

$$Y_0 \boxdot Y_1 = \lambda R.\ \bigcup \{\, \eta_0 \odot Y_1(R \cup \mathsf{R}(\eta_0)) \mid \eta_0 \in Y_0(R)\,\}$$

where, for all histories η, $\mathsf{R}(\eta) \subseteq \mathsf{Res} \cup \{\downarrow\}$ *is defined inductively as follows:*

$$\mathsf{R}(\varepsilon) = \{\downarrow\} \quad \mathsf{R}(\eta\,\alpha(\rho)) = \begin{cases} \mathsf{R}(\eta) \cup \{r\} & \text{if } \rho = r \text{ and } ! \notin \eta \\ \mathsf{R}(\eta) & \text{if } \rho = ? \end{cases} \quad \mathsf{R}(\eta\,!) = \mathsf{R}(\eta) \setminus \{\downarrow\}$$

The denotational semantics $[\![H]\!]_\theta$ of history expressions (Def. 9) is a function from finite sets of resources to the cpo D_0 of sets X of histories such that (i) $! \in X$, and (ii) $\eta ! \in X$ whenever $\eta \in X$. The finite set of resources collects those already used, so making them unavailable for future creations. As usual, the parameter θ binds the free variables of H (in our case, to values in D_0). Note that the semantics is prefix-closed, i.e. for each H and R, the histories in $[\![H]\!](R)$ comprise all the possible truncated prefixes.

Definition 9. Denotational semantics of history expressions

Let D_0 *be the following cpo of sets of histories ordered by set inclusion:* $D_0 = \{\, X \subseteq \mathsf{Ev}^* \cup \mathsf{Ev}^*\,! \mid\ ! \in X \ \wedge\ \forall \eta \in X : \eta ! \in X \,\}$. *The set* $\{\,!\,\}$ *is the bottom element of* D_0. *Let* $D_{den} = \mathcal{P}_{fin}(\mathsf{Res}) \to D_0$ *be the cpo of functions from the finite subsets of* Res *to* D_0. *Note that the bottom element* $\bot$ *of* D_{den} *is* $\lambda R.\,\{\,!\,\}$. *Let* H *be a history expression such that* $fn(H) = \emptyset$, *and let* θ *be a mapping from variables* h *to functions in* D_{den} *such that* $dom(\theta) \supseteq fv(H)$. *The denotational semantics* $[\![H]\!]_\theta$ *is a function in* D_{den}, *inductively defined as follows.*

$$[\![\varepsilon]\!]_\theta = \lambda R.\,\{\,!, \varepsilon\} \qquad [\![\,!\,]\!]_\theta = \bot \qquad [\![h]\!]_\theta = \theta(h) \qquad [\![H \cdot H']\!]_\theta = [\![H]\!]_\theta \boxdot [\![H']\!]_\theta$$

$$[\![\nu n.\,H]\!]_\theta = \lambda R.\ \bigcup_{r \notin R \cup \mathsf{Res}_s} [\![H\{r/n\}]\!]_\theta(R \cup \{r\}) \quad [\![H + H']\!]_\theta = [\![H]\!]_\theta \sqcup [\![H']\!]_\theta$$

$$[\![\alpha(\rho)]\!]_\theta = \lambda R.\,\{\,!, \alpha(\rho), \alpha(\rho)\,!\} \qquad [\![\mu h.H]\!]_\theta = \bigsqcup_{i \geq 0} f^i(\bot) \quad f(Z) = [\![H]\!]_{\theta\{Z/h\}}$$

The first three rules are straightforward. The semantics of $H \cdot H'$ combines the semantics of H and H' with the operator $\boxdot$. The semantics of $\nu n.\,H$ joins the semantics of H, where the parameter R is updated to record the binding of n with r, for *all* the resources r not yet used in R. The semantics of $H + H'$ is the least upper bound of the semantics of H and H'. The semantics of an event comprises the possible truncations. The semantics of a recursion $\mu h.\,H$ is the least upper bound of the ω-chain $f^i(\lambda R.\{\,!\,\})$, where $f(Z) = [\![H]\!]_{\theta\{Z/h\}}$.

We first check that the above semantics is well-defined. First, the image of the semantic function is indeed in D_0: it is easy to prove that, for all H, θ and R, $! \in [\![H]\!]_\theta(R)$ and $\eta ! \in [\![H]\!]_\theta(R)$ whenever $\eta \in [\![H]\!]_\theta(R)$. Lemma B3 [6] guarantees that the least upper bound in the last equation exists (since f is monotone). Also, since f is continuous and $\bot$ is the bottom of the cpo D_{den}, by the Fixed Point theorem the semantics of $\mu h.\,H$ is the least fixed point of f.

Example 3. Consider the following history expressions:

$$H_0 = \mu h.\, \alpha(r) \cdot h \qquad H_1 = \mu h.\, h \cdot \alpha(r) \qquad H_2 = \mu h.\, \nu n.\, (\varepsilon + \alpha(n) \cdot h)$$

Then, $[\![H_0]\!](\emptyset) = \alpha(r)^*!$, i.e. H_0 generates histories with an arbitrary, finite number of $\alpha(r)$. Note that all the histories of H_0 are non-terminating (as indicated by the $!$) since there is no way to exit from the recursion. Instead, $[\![H_1]\!](\emptyset) = \{!\}$, i.e. H_1 loops forever, without generating any events. The semantics of $[\![H_2]\!](\emptyset)$ consists of all the histories of the form $\alpha(r_1) \cdots \alpha(r_k)$ or $\alpha(r_1) \cdots \alpha(r_k)!$, for all $k \geq 0$ and pairwise distinct resources r_i. □

We now define a preorder $H \sqsubseteq H'$ betweeen history expressions, that we shall use in subtyping. Roughly, when $H \sqsubseteq H'$ holds, the histories of H are included in those of H'. The preorder $\sqsubseteq$ includes equivalence, and it is closed under contexts. A history expression H can be arbitrarily "weakened" to $H + H'$. An event $\alpha(\rho)$ can be weakened to $\alpha(?)$, as ? stands for an unknown resource.

Definition 10. Subeffecting

The relation $=$ over history expressions is the least congruence including α-conversion such that the operation $+$ is associative, commutative and idempotent; $\cdot$ is associative, has identity ε, and distributes over $+$, and:

$$\mu h.H = H\{\mu h.\, H/h\} \qquad \mu h.\mu h'.H = \mu h'.\mu h.H \qquad \nu n.\nu n'.H = \nu n'.\nu n.H$$

$$\nu n.\varepsilon = \varepsilon \qquad \nu n.(H + H') = (\nu n.H) + H' \;\; \textit{if } n \notin \textit{fn}(H')$$

$$\nu n.(H \cdot H') = H \cdot (\nu n.H') \;\; \textit{if } n \notin \textit{fn}(H) \qquad \nu n.(H \cdot H') = (\nu n.H) \cdot H' \;\; \textit{if } n \notin \textit{fn}(H')$$

The relation $\sqsubseteq$ over history expressions is the least precongruence such that:

$$H \sqsubseteq H' \;\; \textit{if } H = H' \qquad H \sqsubseteq H + H' \qquad \alpha(\rho) \sqsubseteq \alpha(?)$$

We now formally state that the subeffecting relation agrees with the semantics of history expressions, i.e. it implies trace inclusion. Actually, this turns out to be a weaker notion than set inclusion, because the rule $\alpha(\rho) \sqsubseteq \alpha(?)$ allows for abstracting some resource with a ?. We then render trace inclusion with the preorder $\subseteq_?$ defined below. Intuitively, $\eta \subseteq_? \eta'$ means that η concretizes each unknown resource in η' with some $r \in \mathsf{Res}$.

Definition 11. *The preorder $\subseteq_?$ between histories is inductively defined as:*

$$\varepsilon \subseteq_? \varepsilon \qquad \eta\, \alpha(\rho) \subseteq_? \eta'\, \alpha(\rho') \;\; \textit{if } \eta \subseteq_? \eta' \textit{ and } \rho' \in \{\rho, ?\} \qquad \eta\,! \subseteq_? \eta'\,! \;\; \textit{if } \eta \subseteq_? \eta'$$

The preorder $\subseteq_?$ is extended to sets of histories as follows:

$$I \subseteq_? J \qquad \textit{if } \forall \eta \in I : \exists \eta' \in J : \eta \subseteq_? \eta'$$

The correctness of subeffecting is stated in Lemma 1 below. When $H = H'$ (resp. $H \sqsubseteq H'$), the histories of H are equal to (resp. are $\subseteq_?$ of) those of H'.

Lemma 1. *For all closed history expressions H, H' and for all $R \subseteq \mathsf{Res}$:*

- *if $H = H'$ then $[\![H]\!](R) = [\![H']\!](R)$*
- *if $H \sqsubseteq H'$ then $[\![H]\!](R) \subseteq_? [\![H']\!](R)$.*

4 ν-Types and Type and Effect System

In this section we introduce ν-types, and we use them to define a type and effect system for the calculus of Section 2 (Def. 14). Informally, a term with ν-type $\zeta = \nu N.\, \tau \rhd H$ will have the *pure type* τ, and the *effect* of its evaluation will be a history included in the denotation of the history expression H. The heading νN is used to bind the names $n \in N$ both in τ and H. Pure types comprise:

- the unit type $\mathbf{1}$, inhabited by the value $*$ (and by $\boldsymbol{!}$).
- sets S, to approximate the possible targets of actions. Sets S either contain resources and (possibly) one name, or we have $S = \{?\}$, meaning that the target object is unknown.
- functional types $\tau \to \zeta$. The type ζ is a ν-type, that may comprise the *latent* effect associated with an abstraction.

Example 4. The term $e = (b)\,?\, r : r'$ has type $\{r, r'\} \rhd \varepsilon$ (we omit the νN when $N = \emptyset$). The pure type $\{r, r'\}$ means that e evaluates to either r or r', while producing an empty history (denoted by the history expression ε).

The term $e' = \mathbf{new}\ x\ \mathbf{in}\ \alpha(x); x$ creates a new resource r, fires on it the action α, and then evaluates to r. A suitable type for e' is then $\nu n.\, \{n\} \rhd \alpha(n)$.

The function $g = \lambda_z y.\, \mathbf{new}\ x\ \mathbf{in}\ (\alpha(x); (b)\,?\, x : z\, x)$, instead, has type $\mathbf{1} \to (\{?\} \rhd \mu h.\, \nu n.\, \alpha(n) \cdot (\varepsilon + h)) \rhd \varepsilon$. The latent effect $\mu h.\, \nu n.\, \alpha(n) \cdot (\varepsilon + h)$ records that g is a recursive function that creates a fresh resource upon each recursion step. The type $\{?\}$ says that g will return a resource with unknown identity, since it cannot be predicted when the guard b will become true. $\square$

Type environments are finite mappings from variables and resources to pure types. Roughly, a typing judgment $\Delta \vdash e : \nu N.\, \tau \rhd H$ means that, in a type environment Δ, the term e evaluates to a value of type $\nu N.\, \tau$, and it produces a history represented by $\nu N.\, H$. Note however that the ν-type $\nu N.\, \tau \rhd H$ is more precise than taking $\nu N.\, \tau$ and $\nu N.\, H$ separately. Indeed, in the ν-type the names N indicate exactly the *same* fresh resources in both τ and H.

Definition 12. Types, type environments, and typing judgements

$S ::= R \mid R \cup \{n\} \mid \{?\}$	$R \subseteq \mathsf{Res}, n \in \mathsf{Nam}, S \neq \emptyset$	*resource sets*
$\tau ::= \mathbf{1} \mid S \mid \tau \to \zeta$		*pure types*
$\zeta ::= \nu n.\, \zeta \mid \tau \rhd H$		*ν-types*
$\Delta ::= \emptyset \mid \Delta; r : \{r\} \mid \Delta;\ x : \tau$	$x \notin dom(\Delta)$	*type environments*
$\Delta \vdash e : \zeta$		*typing judgements*

We also introduce the following shorthands (we write $N \pitchfork M$ for $N \cap M = \emptyset$):

$$\nu N.\, \zeta = \nu n_1 \cdots \nu n_k.\, \zeta \qquad \textit{if } N = \{n_1, \ldots n_k\}$$
$$H \cdot \zeta = \nu N.\, \tau \rhd H \cdot H' \qquad \textit{if } \zeta = \nu N.\, \tau \rhd H' \textit{ and } N \pitchfork \textit{fn}(H)$$

*We say $\nu N.\, \tau \rhd H$ is in ν-*normal form *(abbreviated νNF) when $N \subseteq \textit{fn}(\tau)$.*

We now define the subtyping relation $\sqsubseteq$ on ν-types. It builds over the subeffecting relation between history expressions (Def. 10). The first equation in Def. 13 below is a variant of the usual name extrusion. The first two rules for $\sqsubseteq$ allow for weakening a pure type S to a wider one, or to the pure type $\{?\}$. The last rule extends to ν-types the relations $\sqsubseteq$ over pure types and over effects.

Definition 13. Subtypes

The equational theory of types includes that of history expressions (if $H = H'$ then $\tau \rhd H = \tau \rhd H'$), α-conversion of names, and the following equation:

$$\nu n.\, (\tau \rhd H) = \tau \rhd (\nu n.\, H) \quad \textit{if } n \notin \mathit{fn}(\tau)$$

The relation $\sqsubseteq$ over pure types is the least preorder including $=$ such that:

$$S \sqsubseteq S' \;\; \textit{if } S \subseteq S' \textit{ and } S \neq \{?\} \qquad S \sqsubseteq \{?\}$$

$$\nu N.\, \tau \rhd H \sqsubseteq \nu N.\, \tau' \rhd H' \quad \textit{if } \tau \sqsubseteq \tau' \textit{ and } H \sqsubseteq H' \textit{ and } (\mathit{fn}(\tau') \setminus \mathit{fn}(\tau)) \not\pitchfork N$$

Note that the side condition in the last rule above prevents from introducing name captures. For instance, let $\zeta = \nu n.\, \{r\} \rhd \alpha(n)$ and $\zeta' = \nu n.\, \{r, n\} \rhd \alpha(n)$. Since $n \in \mathit{fn}(\{r, n\}) \setminus \mathit{fn}(\{r\})$, then $\zeta \not\sqsubseteq \zeta'$. Indeed, by the equational theory:

$$\zeta \;=\; \{r\} \rhd \nu n.\, \alpha(n) \;=\; \{r\} \rhd \nu n'.\, \alpha(n')$$

After an α-conversion, the subtyping $\zeta \sqsubseteq \zeta'' = \{r, n\} \rhd \nu n'.\, \alpha(n')$ holds. Indeed, in ζ'' the name n' upon which α acts has nothing to do with name n in the pure type $\{r, n\}$, while in ζ' both α and the pure type refer to the same name.

Remark 1. Note that it is always possible to rewrite any type $\nu N.\, \tau \rhd H$ in νNF. To do that, let $\hat{N} = N \cap \mathit{fn}(\tau)$, and let $\check{N} = N \setminus \mathit{fn}(\tau)$. Then, the equational theory of types gives: $\nu N.\, \tau \rhd H = \nu \hat{N}.\, \tau \rhd (\nu \check{N}.H)$.

We now state in Lemma 2 a fundamental result about subtyping of ν-types. Roughly, whenever $\zeta \sqsubseteq \zeta'$, it is possible to α-convert the names of ζ so to separately obtain subtyping between the pure types of ζ and ζ', and subeffecting between their effects. Note that Remark 1 above enables us to use Lemma 2 on any pair of types, after rewriting them in νNF.

Lemma 2. *Let $\nu N.\, \tau \rhd H \sqsubseteq \nu N'.\, \tau' \rhd H'$, where both types are in νNF.*

- *If $\tau' \neq \{?\}$, then there exists a bijective function $\sigma : N \leftrightarrow N'$ such that $\tau\sigma \sqsubseteq \tau'$ and $H\sigma \sqsubseteq H'$.*
- *If $\tau' = \{?\}$, then $\tau \sqsubseteq \tau'$ and $\nu N.H \sqsubseteq H'$.*

Example 5. Let $\zeta = \nu n.\, \{n\} \rhd \alpha(n)$, let $\zeta' = \nu n'.\, \{n', r\} \rhd \alpha(n') + \alpha(r)$, and let $\zeta'' = \{?\} \rhd \nu n''.\, \alpha(n'') + \alpha(?)$. By using Lemma 2 on $\zeta \sqsubseteq \zeta'$, we obtain $\sigma = \{n'/n\}$ such that $\{n\}\sigma \sqsubseteq \{n', r\}$ and $\alpha(n)\sigma \sqsubseteq \alpha(n') + \alpha(r)$. By Lemma 2 on $\zeta' \sqsubseteq \zeta''$, we find $\{n', r\} \sqsubseteq \{?\}$ and $\nu n'.\, \alpha(n') + \alpha(r) \sqsubseteq \nu n''.\, \alpha(n'') + \alpha(?)$. $\square$

Definition 14. Type and effect system

$$\textsc{T-Unit}\ \ \Delta \vdash * : \mathbf{1} \rhd \varepsilon \qquad \textsc{T-Bang}\ \ \Delta \vdash \boldsymbol{!} : \zeta \qquad \textsc{T-Var}\ \ \Delta; \xi : \tau \vdash \xi : \tau \rhd \varepsilon$$

$$\textsc{T-New}\ \ \Delta \vdash \mathbf{new} : \nu n.\, \{n\} \rhd \varepsilon \qquad \textsc{T-Ev}\ \ \Delta; \xi : S \vdash \alpha(\xi) : \mathbf{1} \rhd \textstyle\sum_{\rho \in S} \alpha(\rho)$$

$$\textsc{T-AddVar}\ \frac{\Delta \vdash e : \zeta}{\Delta; \xi : \tau \vdash e : \zeta} \qquad \textsc{T-Abs}\ \frac{\Delta; x : \tau; z : \tau \to \zeta \vdash e : \zeta}{\Delta \vdash \lambda_z x.e : (\tau \to \zeta) \rhd \varepsilon}$$

$$\textsc{T-Wk}\ \frac{\Delta \vdash e : \zeta}{\Delta \vdash e : \zeta'}\ \zeta \sqsubseteq \zeta' \qquad \textsc{T-If}\ \frac{\Delta \vdash e : \zeta \quad \Delta \vdash e' : \zeta}{\Delta \vdash (b)\,?\,e : e' : \zeta}$$

$$\textsc{T-App}\ \frac{\Delta \vdash e : \nu N.(\tau \to \zeta) \rhd H \quad \Delta \vdash e' : \nu N'.(\tau \rhd H')}{\Delta \vdash e\,e' : \nu(N \cup N').\,(H \cdot H' \cdot \zeta)} \quad \begin{array}{l} N \not\pitchfork N' \\ N \not\pitchfork \mathit{fn}(\Delta) \not\pitchfork N' \\ N \not\pitchfork \mathit{fn}(H') \end{array}$$

Here we briefly comment on the most peculiar typing rules.

- (T-Bang) An aborted computation can be given any type, modelling the fact that nothing is known about the behaviour of the term that was aborted.
- (T-New) The type of a **new** is a set $\{n\}$, where n is bound by an outer νn, and the actual effect is empty. (We could instead record the resource creation in the effect, by handling **new** as we currently do for $(\lambda x.\, \alpha_{\mathsf{created}}(x); x)\mathbf{new}$.)
- (T-Ev) An event $\alpha(\xi)$ has type $\mathbf{1}$, provided that the type of ξ is a set S. The effect of $\alpha(\xi)$ can be any of the accesses $\alpha(\rho)$ for ρ included in S.
- (T-Abs) The actual effect of an abstraction is the empty history expression, while the latent effect (included in the type ζ) is equal to the actual effect of the function body. Note that ζ occurs twice in the premise: to unify those occurrences, usually one has to resort to recursive history expressions $\mu h.\, H$.
- (T-Wk) This rule allows for *weakening* of ν-types, according to Def. 13.
- (T-App) The effects in the rule for application are concatenated according to the evaluation order of the call-by-value semantics (function, argument, latent effect). The side conditions ensure that there is no clash of names. In particular, the disjointness condition makes sure that the names created by the function are never used by the argument.

Example 6. We have the following typing judgements, in the (omitted) empty typing environment (detailed typing derivations can be found in [6]):

$$\vdash e_1 = (\,b\,)\,?\ \lambda_z x.\,\alpha\ :\ \lambda_z x.\,\beta : (\mathbf{1} \to (\mathbf{1} \rhd \alpha + \beta)) \rhd \varepsilon$$

$$\vdash e_2 = \lambda_g x.\,(b')\,?\,* : g(e_1\,x) : (\mathbf{1} \to (\mathbf{1} \rhd \mu h.\,\varepsilon + (\alpha + \beta) \cdot h)) \rhd \varepsilon$$

$$\vdash e_3 = \alpha(\mathbf{new}\ x\ \mathbf{in}\ (b)\,?\,x : r) : \mathbf{1} \rhd \nu n.\,(\alpha(n) + \alpha(r))$$

$$\begin{aligned} \vdash e_4 &= \mathbf{let}\ f = (\lambda x.\,\mathbf{new}\ y\ \mathbf{in}\ \alpha(y); y)\ \mathbf{in}\ \beta(f*; f*) \\ &\quad : \mathbf{1} \rhd (\nu n.\,\alpha(n)) \cdot (\nu n'.\,\alpha(n') \cdot \beta(n')) \end{aligned}$$

$$\vdash e_5 = \mathbf{let}\ g = (\mathbf{new}\ y\ \mathbf{in}\ \lambda x.\,\alpha(y); y)\ \mathbf{in}\ \beta(g*; g*) : \mathbf{1} \rhd \nu n.\,\alpha(n) \cdot \alpha(n) \cdot \beta(n)$$

$$\vdash e_6 = (\lambda_z x.\,\mathbf{new}\ y\ \mathbf{in}\ (b)\,?\,\alpha(y) : \beta(y); zx)\,* : \mathbf{1} \rhd \mu h.\,\nu n.\,(\alpha(n) + \beta(n) \cdot h)$$

$$\vdash e_7 = \alpha((\lambda_z x.\,\mathbf{new}\ y\ \mathbf{in}\ (b)\,?\,y : \beta(y); zx)\,*) : \mathbf{1} \rhd (\mu h.\,\nu n.\,(\varepsilon + \beta(n) \cdot h)) \cdot \alpha(?)$$

The effects of e_4 and e_5 correctly represent the fact that two distinct resources are generated by e_4, while the evaluation of e_5 creates a single fresh resource. The effect of e_6 is a recursion, at each step of which a fresh resource is generated. The effect of e_7 is more peculiar: it behaves similarly to e_6 until the recursion is left, when the last generated resource is exported. Since its identity is lost, the event α is fired on the unknown resource "?". □

The following lemma relates the histories denoted by a context C with the typing of any term of the form $C[v]$. More precisely, the histories of C are included (modulo concretization of ?) in those denoted by the effect in the ν-type. Since the big-step semantics of terms produces both a value v and a context C, this result will be pivotal in proving the correctness of our type and effect system.

Lemma 3. *For all closed history contexts C, values v, and sets of resources R:*

$$\Delta \vdash C[v] : \nu N.\, \tau \rhd H \implies \mathcal{H}(C, R) \subseteq_? [\![\nu N.\, H]\!](R)$$

We now establish a fundamental result about typing, upon which the proof of the Subject Reduction lemma is based. Roughly, given a history context C and a term e, it allows for constructing a type for $C[e]$ from a type for e, and *viceversa*. The information needed to extend/reduce a type is contained in $\mathcal{T}(C, \Delta)$, that extracts from C a set of binders, a history expression, and a type environment.

Definition 15. *For all C and Δ, we inductively define $\mathcal{T}(C, \Delta)$ as follows:*

$$\mathcal{T}(\bullet, \Delta) = (\varepsilon, \emptyset)$$

$$\mathcal{T}(\alpha(\xi); C', \Delta) = (\textstyle\sum_{\rho \in \Delta(\xi)} \alpha(\rho) \cdot H', \Delta') \text{ if } \mathcal{T}(C', \Delta) = (H', \Delta')$$

$$\mathcal{T}(\textbf{new } x \textbf{ in } C', \Delta) = (\nu n.H', \Delta'; x : \{n\}) \text{ if } \mathcal{T}(C', \Delta; x : \{n\}) = (H', \Delta'), n \notin \Delta$$

Hereafter, when writing $\mathcal{T}(C, \Delta) = (\nu N.\, H, \Delta')$ we always assume $N = fn(\Delta')$. This is always possible by the equational theory of history expressions (Def. 10).

Lemma 4. *Let $\mathcal{T}(C, \Delta) = (\nu N.\, H, \Delta')$. Then, for all terms e:*

- $\Delta; \Delta' \vdash e : \zeta' \implies \Delta \vdash C[e] : \nu N.\, H \cdot \zeta'$
- $\Delta \vdash C[e] : \zeta \implies \exists \zeta' : \Delta; \Delta' \vdash e : \zeta'$ *and* $\nu N.\, H \cdot \zeta' \sqsubseteq \zeta$

We state below the Subject Reduction Lemma, crucial for proving our type and effect system correct. We state it in the traditional form where the type is preserved under computations. This was made possible by the big-step semantics of terms, where all the information about the generated histories is kept in a history context. Note instead this were not the case for a small-step operational semantics, like the one in [3], where histories grow along with computations. This would require Subject Reduction to "consume" the target type, to render the events fired, and the resources created, in execution steps. Not preserving the type would make the inductive statement harder to to write and to prove.

Lemma 5 (Subject Reduction). *If $\Delta \vdash e : \zeta$ and $e \overset{C}{\Longrightarrow} v$, then $\Delta \vdash C[v] : \zeta$.*

Theorem 1 below guarantees that our type and effect system correctly approximates the dynamic semantics, i.e. the effect of a term e represents all the possible run-time histories of e. As usual, precision is lost with conditionals and with recursive functions. Also, you may lose the identity of names exported by recursive functions (see e.g. the type of e_7 in Ex. 6).

Theorem 1 (Correctness of effects). *For all closed terms e:*

$$\Delta \vdash e : \nu N.\, \tau \rhd H \implies \mathcal{H}(e) \subseteq_? [\![\nu N.\, H]\!](\emptyset)$$

Proof. By Def. 5, $\mathcal{H}(e) = \bigcup_{e \xRightarrow{C} v} \mathcal{H}(C, \emptyset)$. Let C and v be such that $e \xRightarrow{C} v$. By Lemma 5, $\Delta \vdash C[v] : \nu N.\, \tau \rhd H$. By Lemma 3, $\mathcal{H}(C, \emptyset) \subseteq_? [\![\nu N.\, H]\!](\emptyset)$. Therefore, $\mathcal{H}(e) \subseteq_? [\![\nu N.\, H]\!](\emptyset)$. □

5 Conclusions

We studied how to correctly and precisely record creation and use of resources in a type and effect system for an extended λ-calculus. To do that, we used the ν-quantifier for denoting freshness in types and effects. The main technical result is Theorem 1, which guarantees the type of a program correctly approximates its run-time histories. This enables us to exploit the model-checking technique of [2] to verify history-based usage policies of higher-order programs.

Future Work. To improve the accuracy of types, we plan to relax the constraint that a single name can appear in pure types S. For instance, consider the term:

$$e = \textbf{new}\ x\ \textbf{in}\ \textbf{new}\ y\ \textbf{in}\ (\beta(x); \beta(y); (b)\,?\,x : y))$$

Currently, we have the judgements $\vdash e : \{?\} \rhd \nu n.\, \nu n'.\, \beta(n) \cdot \beta(n')$, and thus $\vdash \alpha(e) : \mathbf{1} \rhd \nu n.\, \nu n'.\, \beta(n) \cdot \beta(n') \cdot \alpha(?)$ whereas by relaxing the single-name assumption on pure types S, we would have the more precise judgements $\vdash e : \nu\{n, n'\}.\, \{n, n'\} \rhd \beta(n) \cdot \beta(n')$ and $\vdash \alpha(e) : \mathbf{1} \rhd \nu n. \nu n'.\, \beta(n) \cdot \beta(n') \cdot (\alpha(n) + \alpha(n'))$.

A further improvement would come from allowing subtyping of functional types, e.g. by extending Def. 13 with the rule $\tau \to \zeta \sqsubseteq \tau' \to \zeta'$ if $\tau' \sqsubseteq \tau$ and $\zeta \sqsubseteq \zeta'$ (i.e. contravariant in the argument and covariant in the result). Let e.g. $f = \lambda x.\, ((b)\,?\,\lambda.\, \alpha : x); x$. With the current definition, we have $\vdash f\,(\lambda.\, \beta) : (\mathbf{1} \to (\mathbf{1} \rhd \alpha + \beta)) \rhd \varepsilon$. Note that the function $\lambda.\, \alpha$ is discarded, and so we would like to have instead $\vdash f\,(\lambda.\, \beta) : (\mathbf{1} \to (\mathbf{1} \rhd \beta)) \rhd \varepsilon$, which is more accurate. Subtyping of functional types would allow for such a judgement, using the weakening $\mathbf{1} \to (\mathbf{1} \rhd \beta) \sqsubseteq \mathbf{1} \to (\mathbf{1} \rhd \alpha + \beta)$ within the typing judgement of f.

The above constraints have been introduced in our model in order to simplify the proofs, only (for instance, the restriction about the number of names in set types helps in the proof of Lemma B20 [6]). Even when exploiting these constraints, the technical burden in our proofs is still quite heavy: yet, we conjecture that these restrictions could be lifted without invalidating our main results.

We plan to develop a type and effect inference algorithm, taking [19] as a starting point. The subtype relation of [19] enjoys some nice properties, e.g. principal types, which we expect to maintain in our setting. The main difference is that, while [19] constructs and resolves separately type constraints and effect constraints, ours demands for dealing with subtyping constraints between whole ν-types. The key issue is unifying α-convertible terms, which we expect to manage by exploiting nominal unification [21].

Acknowledgements. This work has been partially supported by EU-FETPI Global Computing Project IST-2005-16004 SENSORIA (Software Engineering for Service-Oriented Overlay Computers) and by the MIUR-PRIN project SOFT (Tecniche Formali Orientate alla Sicurezza).

References

1. Bartoletti, M., Degano, P., Ferrari, G.L.: History based access control with local policies. In: Sassone, V. (ed.) FOSSACS 2005. LNCS, vol. 3441, pp. 316–332. Springer, Heidelberg (2005)
2. Bartoletti, M., Degano, P., Ferrari, G.L., Zunino, R.: Local policies for resource usage analysis. To appear in ACM Tran. Progr. Lang. and Sys.
3. Bartoletti, M., Degano, P., Ferrari, G.L., Zunino, R.: Types and effects for resource usage analysis. In: Seidl, H. (ed.) FOSSACS 2007. LNCS, vol. 4423, pp. 32–47. Springer, Heidelberg (2007)
4. Bartoletti, M., Degano, P., Ferrari, G.L., Zunino, R.: Hard life with weak binders. In: Proc. EXPRESS (2008)
5. Bartoletti, M., Degano, P., Ferrari, G.L., Zunino, R.: Model checking usage policies. In: Proc. Trustworthy Global Computing (2008)
6. Bartoletti, M., Degano, P., Ferrari, G.L., Zunino, R.: ν-types for effects and freshness analysis. Technical Report DISI-09-033, DISI - Università degli Studi di Trento (2009)
7. Bergstra, J.A., Klop, J.W.: Algebra of communicating processes with abstraction. Theoretical Computer Science 37 (1985)
8. Bradfield, J.: On the expressivity of the modal μ-calculus. In: Puech, C., Reischuk, R. (eds.) STACS 1996. LNCS, vol. 1046. Springer, Heidelberg (1996)
9. Chaki, S., Rajamani, S.K., Rehof, J.: Types as models: model checking message-passing programs. In: Proc. POPL (2002)
10. Esparza, J.: On the decidability of model checking for several μ-calculi and Petri nets. In: Tison, S. (ed.) CAAP 1994. LNCS, vol. 787. Springer, Heidelberg (1994)
11. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. In: Proc. POPL (2008)
12. Igarashi, A., Kobayashi, N.: Resource usage analysis. In: Proc. POPL (2002)
13. Igarashi, A., Kobayashi, N.: A generic type system for the pi-calculus. Theoretical Computer Science 311(1-3) (2004)
14. Milner, R., Parrow, J., Walker, D.: A Calculus of Mobile Processes, I and II. Information and Computation 100(1) (September 1992)
15. Nielson, H.R., Nielson, F.: Higher-order concurrent programs with finite communication topology. In: Proc. POPL (1994)
16. Odersky, M.: A functional theory of local names. In: Proc. POPL (1994)
17. Shinwell, M.R., Pitts, A.M., Gabbay, M.: FreshML: programming with binders made simple. In: Proc. ICFP (2003)
18. Skalka, C., Smith, S.: History effects and verification. In: Chin, W.-N. (ed.) APLAS 2004. LNCS, vol. 3302, pp. 107–128. Springer, Heidelberg (2004)
19. Skalka, C., Smith, S., Horn, D.V.: Types and trace effects of higher order programs. Journal of Functional Programming 18(2) (2008)
20. Talpin, J.-P., Jouvelot, P.: Polymorphic type, region and effect inference. Journal of Functional Programming 2(3) (1992)
21. Urban, C., Pitts, A.M., Gabbay, M.: Nominal unification. Theoretical Compututer Science 323(1-3) (2004)